



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

2012-03

USB Support for the Least Privilege Separation Kernel

Tan, Hwee Meng

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/6878>

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

USB SUPPORT FOR THE LEAST PRIVILEGE SEPARATION KERNEL

by

Hwee Meng Tan

March 2012

Thesis Advisor:
Co-Advisor:

Paul C. Clark
Cynthia E. Irvine

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2012	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE USB Support for the Least Privilege Separation Kernel			5. FUNDING NUMBERS	
6. AUTHOR(S) Hwee Meng Tan				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number N/A.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) The objective of the Trusted Computing Exemplar (TCX) project is to construct a useful example of a high assurance component as a reference model for building high assurance systems. The targeted exemplar component is the Least Privilege Separation Kernel (LPSK). The LPSK currently exists in the form of a partially implemented prototype. With respect to support for external computer peripherals, the prototype LPSK currently supports visual display units (e.g., LCD monitors) for output and PS/2 keyboards for user inputs. The PS/2 interface is a technology that is fast disappearing from computer systems. To keep the support for external computer peripherals up to date, support for other interface standards such as USB is required. This work has identified and ported part of the USB implementation (i.e., USB bus driver, UHCI driver and hub driver) from an open source BIOS called Sea BIOS to the LPSK. The problems encountered in the course of porting, together with their solutions, are presented to facilitate future work. Tests were conducted to ensure correct behavior of the ported code.				
14. SUBJECT TERMS TCX project, LPSK, USB standards, HID class driver and PCI BUS			15. NUMBER OF PAGES 85	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

USB SUPPORT FOR THE LEAST PRIVILEGE SEPARATION KERNEL

Hwee Meng Tan
Civilian, Defence Science & Technology Agency, Singapore
B.Eng., Nanyang Technological University, Singapore, 1999
MTech, Institute of System Sciences, National University of Singapore, Singapore, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
March 2012**

Author: Hwee Meng Tan

Approved by: Paul C. Clark
Thesis Advisor

Cynthia E. Irvine
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The objective of the Trusted Computing Exemplar (TCX) project is to construct a useful example of a high assurance component as a reference model for building high assurance systems. The targeted exemplar component is the Least Privilege Separation Kernel (LPSK). The LPSK currently exists in the form of a partially implemented prototype. With respect to support for external computer peripherals, the prototype LPSK currently supports visual display units (e.g., LCD monitors) for output and PS/2 keyboards for user inputs. The PS/2 interface is a technology that is fast disappearing from computer systems. To keep the support for external computer peripherals up to date, support for other interface standards such as USB is required. This work has identified and ported part of the USB implementation (i.e., USB bus driver, UHCI driver and hub driver) from an open source BIOS called Sea BIOS to the LPSK. The problems encountered in the course of porting, together with their solutions, are presented to facilitate future work. Tests were conducted to ensure correct behavior of the ported code.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	MOTIVATION	1
B.	PURPOSE OF STUDY.....	1
C.	THESIS ORGANIZATION.....	2
II.	BACKGROUND	3
A.	LEAST PRIVILEGE SEPARATION KERNEL (LPSK).....	3
B.	TRUSTED COMPUTING EXEMPLAR (TCX)	6
C.	PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS.....	7
D.	UNIVERSAL SERIAL BUS (USB) STANDARDS	8
E.	USB HOST CONTROLLER INTERFACE.....	11
F.	USB CLASS CODES	11
G.	SUMMARY	12
III.	USB IMPLEMENTATION ISSUES FOR THE LPSK	13
A.	SYSTEM STARTUP PROCESS.....	13
B.	LEGACY USB SUPPORT IN BIOS.....	14
C.	PCI LOGIC FLOW	15
D.	UHCI OVERVIEW	18
E.	OPEN SOURCE IMPLEMENTATIONS	21
F.	SUMMARY	23
IV.	PORTING.....	25
A.	DEVELOPMENT ENVIRONMENT	25
B.	LPSK MODIFICATIONS FOR USB SUPPORT.....	26
C.	SEABIOS USB IMPLEMENTATION	27
D.	PROBLEM RESOLUTION.....	28
E.	SUMMARY	33
V.	TESTING.....	35
A.	TEST SETUP	35
B.	TEST PROCEDURES.....	36
C.	TEST RESULTS.....	42
D.	SUMMARY	51
VI.	CONCLUSION	53
A.	RESULTS	53
B.	RECOMMENDATIONS FOR FUTURE WORK.....	53
APPENDIX A.	USB CLASS CODE	55
APPENDIX B.	USB DEVICE INFORMATION FOR TESTING	57
	LIST OF REFERENCES.....	65
	INITIAL DISTRIBUTION LIST	67

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1.	Layers within a typical computer system.....	3
Figure 2.	An example of separation kernel configuration.....	5
Figure 3.	An example of Least Privilege Separation Kernel information flow	6
Figure 4.	Devices connected in a Bus topology	10
Figure 5.	Devices connected in a Tiered topology using hubs.....	10
Figure 6.	UHCI data structure [From 17].....	19
Figure 7.	Show all USB devices.....	40
Figure 8.	Connect (Disconnect from host)	41

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1.	Comparison of speed in different USB versions.....	9
Table 2.	Effect of disabling “USB legacy support” in BIOS.....	15
Table 3.	PCI configuration registers for header type 0x00 [From 16].....	16
Table 4.	PCI 32-bit configuration address	17
Table 5.	Prog IF field in PCI configuration registers.....	17
Table 6.	USB data transfer types	18
Table 7.	UHCI I/O registers.....	20
Table 8.	USB supports in open source OSes evaluated	23
Table 9.	Testing configuration with VMware.....	35
Table 10.	Testing configuration without VMware.....	36
Table 11.	Test result for Keyboard 1	44
Table 12.	Test result for Keyboard 2	46
Table 13.	Test result for Mouse 1	48
Table 14.	Test result for Mouse 2	50
Table 15.	USB class code	55
Table 16.	Reference result for Keyboard 1	58
Table 17.	Reference result for Keyboard 2	60
Table 18.	Reference result for Mouse 1	62
Table 19.	Reference result for Mouse 2.....	64

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
BCD	Binary Coded Decimal
BIOS	Basic Input-Output System
CPU	Central Processing Unit
EAL	Evaluation Assurance Level
EHCI	Enhanced Host Controller Interface
GCC	GNU Compiler Collection
GRUB	Grand Unified Bootloader
HCI	Human Computer Interface
HID	Human Interface Device
KVM	Kernel-based Virtual Machine
LCD	Liquid Crystal Display
LPSK	Least Privilege Separation Kernel
MBR	Master Boot Record
OHCI	Open Host Controller Interface
OO	Object-Orientated
OS	Operating System
PC	Personal Computer
PCI	Peripheral Component Interconnect
PnP	Plug-and-Play
PoLP	Principle of Least Privilege
POST	Power-On Self-Test
PS/2	Personal System/2

QH	Queue Head
SKPP	Separation Kernel Protection Profile
TCX	Trusted Computer Exemplar
TD	Transfer Descriptor
UHCI	Universal Host Controller Interface
USB	Universal Serial Bus
xHCI	Extensible Host Controller Interface

ACKNOWLEDGMENTS

I would like to thank my Singapore sponsor, the Defence Science & Technology Agency (DSTA), for giving me the privilege of furthering my education at the Naval Postgraduate School (NPS). I would also like to express my gratitude to my thesis advisor, Paul Clark, and co-advisor, Professor Cynthia Irvine, for their patience in making this thesis possible. Their guidance provided in the course of this thesis is instrumental in the outcome achieved. The experience and knowledge I gained while in NPS will be an important part of my lifelong learning.

Last but not least, I am grateful to my wife, Lynn, for her patience and support, and to my son for being a key motivator in my life.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The objective of the Trusted Computing Exemplar (TCX) project is to construct a useful example of a high assurance component as a reference model for building high assurance systems. The targeted exemplar component is the Least Privilege Separation Kernel (LPSK). The LPSK currently exists in the form of a partially implemented prototype. With respect to user I/O, the prototype LPSK currently supports the PS/2 keyboard, a technology which is fast disappearing from computer systems. The LPSK will not be useful if it cannot provide adequate human computer interface (HCI) for inputs and outputs. Minimally, to ensure usability in the near future, support for non-PS/2 keyboards is required.

The motivation of this thesis is to develop the Universal Serial Bus (USB) support in the LPSK as part of the continual effort to enhance the LPSK. With USB support, a variety of USB devices could be used with the LPSK. Not only will this address the near term problem of PS/2 keyboards and mice becoming obsolete, in the long run, it will allow LPSK support for other USB devices such as hard disks and printers after basic low-level USB functionality is provided.

B. PURPOSE OF STUDY

Prior to this thesis, there had been no effort to provide USB support in the LPSK. In this thesis, the primary objective was to provide basic USB support such that drivers for USB devices could easily be added to the LPSK. It was determined that it would be easier to provide such functionality by porting an existing implementation instead of writing one from scratch. Therefore, another objective arose to document the lessons learned from the porting effort for reference in future work to enhance the USB implementation in the LPSK.

C. THESIS ORGANIZATION

The subsequent chapters in this thesis are organized as follows: **Chapter II** (Background) provides an overview of the technologies and the background information necessary to understand the work presented in this thesis. Specifically, background information on the LPSK is provided, together with an overview of related technologies fundamental to USB implementation.

Chapter III (USB Implementation issues for the LPSK) highlights the various system level issues that need to be considered when implementing USB support for the LPSK. A detailed walkthrough is provided for how USB controllers and USB devices are detected and configured. In addition, results from the evaluation of three open source USB implementations (Quest OS, Visual OS and Sea BIOS) that are assessed to be potential candidates for porting to the LPSK are presented in this chapter.

Chapter IV (Porting) provides the details for how a subset of a USB driver from the selected open source USB implementation was ported to the LPSK. Specifically, it describes the technical details related to how the parts of the USB modules in Sea BIOS OS were ported to the LPSK.

Chapter V (Testing) covers the testing conducted on the USB code ported from Sea BIOS to the LPSK. It details the scope of testing performed on the modified LPSK and provides the details required to replicate the tests when necessary.

Chapter VI (Conclusion) summarizes the thesis results and provides some recommendations on possible enhancement that can be undertaken as future work for the USB modules in the LPSK.

II. BACKGROUND

The objective of this research is to extend the range of external peripheral devices available for use in the Least Privilege Separation Kernel (LPSK) by providing Universal Serial Bus (USB) support. This chapter is intended to provide the reader with an overview of the technologies and background information necessary to understand the context of the research.

A. LEAST PRIVILEGE SEPARATION KERNEL (LPSK)

To understand the Least Privilege Separation Kernel (LPSK), it is first necessary to understand the concepts and terminology used in the acronym: LPSK. A *kernel* in computing terminology refers to the component of the Operating System (OS) responsible for managing the computer's hardware resources, such as its Central Processing Unit (CPU), memory and devices. An application can request services from an OS through system calls via the OS-specific Application Programming Interfaces (APIs). This may include requests for hardware-related services, which will be passed on to the kernel for processing. A graphical illustration for a typical kernel configuration in a computer system is as shown in the Figure 1.

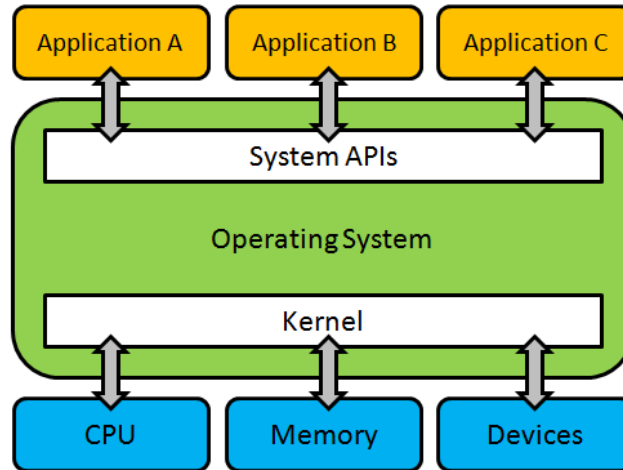


Figure 1. Layers within a typical computer system

A *security kernel* is an isolated and protected portion of a computer system that is intended to enforce the security policy of the system. It implements the reference monitor abstraction [1] which must satisfy the following three fundamental principles [2]:

- **Completeness:** it must be impossible to bypass.
- **Isolation:** it must be tamperproof.
- **Verifiability:** it must be built in such a way that its correctness can be shown.

A variant of the security kernel known as the *separation kernel* was conceptualized and first described by Rushby [3] in 1984. Rushby posited that the security possible in physically distributed systems could be achieved using logical separation. He claimed that a separation kernel should be able to create the same secure environment found in physically distributed systems in a single shared machine. In his paper, Rushby also proposed the separation kernel as a possible solution for addressing the difficulties and problems faced in the development and verification of large, complex security kernels.

Starting after 2000, secure system architects began to explore the use of separation kernels in high assurance systems. To address this growing level of interest, the Information Assurance Directorate of National Security Agency (NSA) in the United States published the Separation Kernel Protection Profile (SKPP) in 2007 [4]. The SKPP defines the security functional and assurance requirements for the construction and evaluation of separation kernels and was written within the framework established by the Common Criteria [5].

In a separation kernel, all subjects (e.g., processes) and resources in a system are partitioned into security policy-equivalence classes. Security of the system is maintained by controlling information flow between the different partitions. A subject in one partition cannot affect entities in another partition (either directly or indirectly) unless it is allowed to as defined by the security policy of the system [6]. A separation kernel is usually implemented in a very small piece of software that can be verified to implement its specified behavior (and no other behavior) and is trusted to enforce logical separation between the different partitions. In each of the logical partitions, middleware that offers

most of the traditional operating system functionality, such as a network stack or file system, can be layered on top of the separation kernel. A graphical illustration of a hypothetical separation kernel configuration with three different partitions is shown in Figure 2.

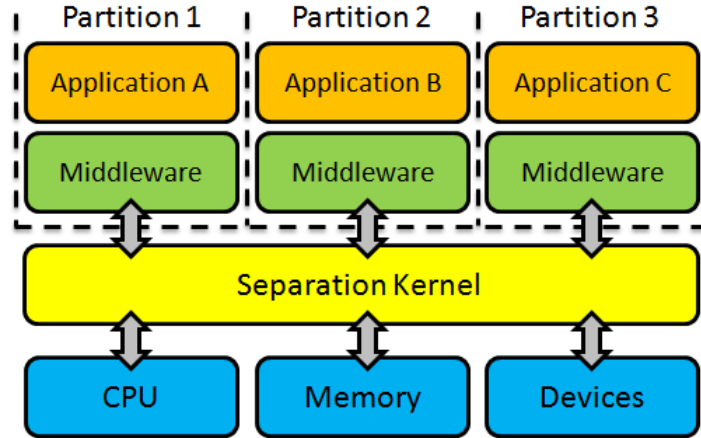


Figure 2. An example of separation kernel configuration

The *Principle of Least Privilege* is an important concept in computer security. Saltzer and Schroeder [7] defined least privilege by stating, “every program and every user of the system should operate using the least set of privileges necessary to complete the job.” When this principle is applied to processes on the computer; it implies that each subject should be accorded with the minimum privilege required for it to access resources necessary to perform its duties. By applying this principle, the likelihood of the computer being exploited and resulting in compromise of security will be greatly reduced.

When a partition contains more than one subject, the separation kernel applies its security policies equally to all subjects within a partition, even if a subject does not need all the given privileges, thus violating the principle of least privilege. A variant of the separation kernel that provides for the principle of least privilege is known as a Least Privilege Separation Kernel (LPSK) [8]. In a LPSK, besides controlling the partition-to-partition information flow similar to a separation kernel, there is also the ability to manage the resources that subjects can access. To illustrate this, consider the example as shown in Figure 3. The system has three partitions, namely 1, 2 and 3. If there is a need

for subject “B” in partition 1 to read from resource “4” in partition 2, the interaction can be restricted to that particular flow instead of allowing all subjects in partition 1 to access all resources in partition 2.

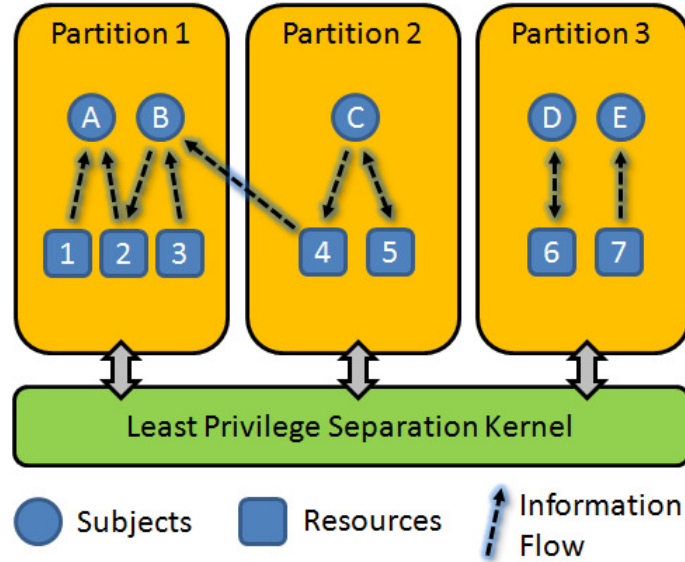


Figure 3. An example of Least Privilege Separation Kernel information flow

B. TRUSTED COMPUTING EXEMPLAR (TCX)

The TCX project [9] is an on-going effort undertaken by the Center for Information Systems Security Studies and Research (CISR) at the U.S. Naval Postgraduate School (NPS). The goal of the TCX project is to provide a working reference implementation of a high assurance trusted computing component that can be openly distributed and shared. The work is motivated by the need for more high assurance products, so a major goal of the project is to demonstrate the processes that must be followed to achieve high assurance. One of the key deliverables in the TCX project is a Least Privilege Separation Kernel (LPSK).

The TCX project consists of four related activities to achieve its goal. These activities are:

- Defining a reusable high assurance development framework,

- Implementing a reference implementation for a trusted network component using the framework defined.
- Providing assurance of the security and assurance of the trusted component by designing and building it to meet the most stringent Evaluation Assurance Level (i.e., EAL7) requirements as defined in Common Criteria [5] as well as to meeting the requirements of the Separation Kernel Protection Profile (SKPP) [4] published by NSA.
- Promulgating the knowledge and know-how by making all source code and documentation developed through the TCX project openly available to the public.

A part of the TCX project is the development of a LPSK. This LPSK is the target of the implementation activity in this thesis. The design and implementation of the LPSK is currently ongoing and the work on incorporating USB support is part of the ongoing effort to improve the prototype LPSK.

C. PERIPHERAL COMPONENT INTERCONNECT (PCI) BUS

PCI stands for *Peripheral Component Interconnect* and is an I/O bus designed to enhance flexibility in small systems. The first version (1.0) of the PCI specification was released in 1992, while the latest version (3.0) was released in 2002. PCI, as the name suggests, helps to extend the functionality of the system by providing a way to easily connect internally installed computer peripherals. In this thesis, the computer peripherals shall be termed generically as “devices.”

To better understand the PCI bus, we will need to understand how components within a typical personal computer (PC) communicate. For components to send and receive information, they must be connected. The interconnections between the components are known collectively as a bus. Modern systems usually have two or more buses to cater to different types of traffic.

A typical PC will have two main buses: a system bus for fast data transfer between the CPU and the system memory and a slower bus for communicating with

devices such as hard disks and sound cards. For the latter, one common bus in current use is the PCI bus. To connect the slower buses used for devices to the system bus, bridges are used to translate the signaling used by the different buses [10].

The PCI bus supports many types of PCI devices. A comprehensive list of these device types can be found at the PCI database website [11] which was originally created by Jim Boemler. Among these PCI device types can be a USB controller which extends the system functionality to support USB devices such as keyboards and mice.

Because the PCI specifications are referenced by many companies to develop PCI compliant devices, the PCI Special Interest Group (PCI-SIG) [12] was formed to maintain and publish all information related to PCI specifications.

D. UNIVERSAL SERIAL BUS (USB) STANDARDS

USB stands for *Universal Serial Bus* and is an I/O standard that was designed to provide a common interface for connecting external devices to a system, in an effort to simplify the common practice of connecting and disconnecting devices. The first official release (Version 1.0 Release Candidate) was made available in 1995. Over the years, newer versions of the USB specifications have been published to address the need to support devices that operate at higher speeds. The latest version is 3.0, which was released in 2008. The maximum transfer rate (speed) for the different USB versions are listed in Table 1.

Version	Also Known As	Speed
1.0	Low Speed	1.5 Mbit/s
1.1	Full Speed	12 Mbit/s
2.0	Hi Speed	480 Mbit/s
3.0	Super Speed	5 Gbit/s

Table 1. Comparison of speed in different USB versions

To achieve the goal of simplifying the plugging of devices to a system, there are some key functional requirements that need to be satisfied. These requirements are:

- **Plug-and-Play (PnP)** capability for detecting newly connected devices without the need to restart the system
- **Self-powering** of low-power consumption devices without the need for an additional power supply
- **Generic device drivers** for popular devices to minimize the need for developing new software drivers.
- **Standardized common interface socket** to prevent proprietary implementations by hardware manufacturers.

Contrary to the name, USB is technically not a bus. It is a serial interface standard because you can connect two or more USB devices in serial, in contrast with the PCI bus where all the PCI devices are connected in a bus topology as shown in Figure 4.

Bus Topology

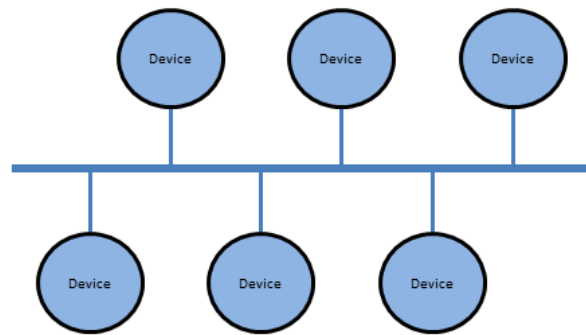


Figure 4. Devices connected in a Bus topology

USB devices are connected in a tiered topology via the use of hubs as shown in Figure 5. At the root of the tier, there is a root hub which is connected to the host (i.e., PC) via the PCI bus. On paper, the USB standard supports up to a total of 127 devices and hubs. In practice, the maximum number of supported devices and hubs is usually much lower due to limitations in speed and power.

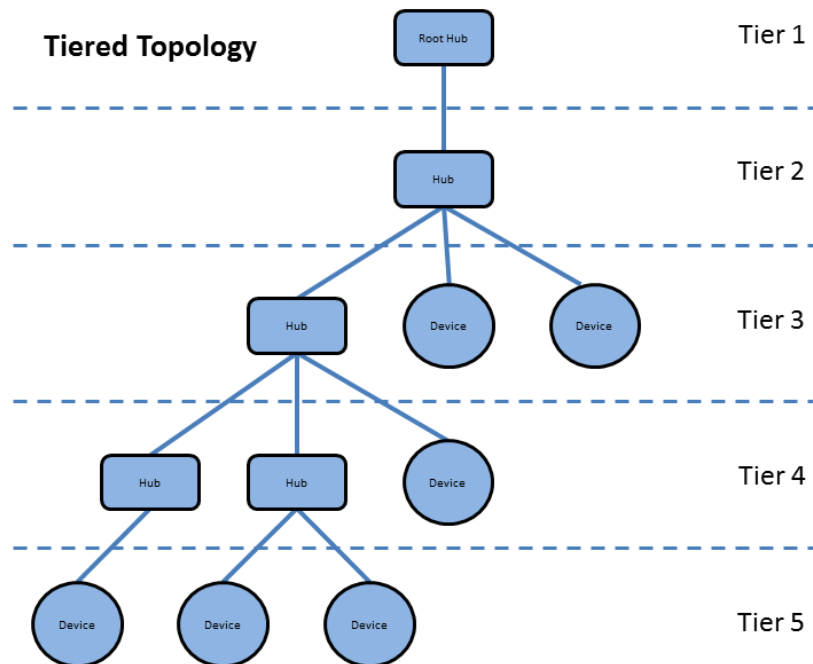


Figure 5. Devices connected in a Tiered topology using hubs

E. USB HOST CONTROLLER INTERFACE

The USB specification includes details on all aspects of implementing the USB standard between a USB Host Controller and connected USB devices, such as the mechanical, electrical and protocol specifications. A USB specification describes the creation of USB packets and the synchronization of communication between the Host Controller and USB devices.

USB device drivers need to communicate with the Host Controller hardware through a register-level interface called Host Controller Interface (HCI), which is separate from the USB specification. There are various HCI specifications published for USB implementations. For USB version 1.0 and 1.1, they are supported by two different HCI specifications. These two specifications are known as the Universal Host Controller Interface (UHCI) and Open Host Controller Interface (OHCI) and they differ in the definition of what is being handled by the USB controllers. The UHCI specification published by Intel is more software driven compared to the OHCI specification published by Compaq. Therefore, UHCI implementations are more CPU intensive but less costly to implement.

For implementing Version 2.0 of the USB standards, the Enhanced Host Controller Interface (EHCI) specification was created by Intel [13] to address the high speed requirement for faster devices. In addition, to ensure backward compatibility with Versions 1.0 and 1.1, Version 2.0 of the USB standards specifies that both UHCI and OHCI specifications must be supported.

The latest version of USB standards is Version 3.0. To support the implementation of this standard, Intel published the Extensible Host Controller Interface (xHCI).

F. USB CLASS CODES

The functionalities provided by the USB devices are determined by its class codes (see Appendix A for a complete list of class codes). When a system is booted, or when a device is plugged into a host (e.g., PC), the system software (e.g., the operating system) will query the USB controller for information about the devices that are connected. The

class codes that are returned by the devices will allow the system software to determine and load the respective software drivers necessary to communicate with each of these devices.

For testing purposes, the class of USB devices that are of interest to this thesis are the keyboards and mice. Both of these devices fall under the HID class. HID stands for *Human Interface Device* and the specification for this class of USB devices can be found in the document “Device Class Definition for HID 1.11” [14]. The HID class of USB devices is used by humans to control the operation of a computer system. In contrast to other classes of devices that require specialized unique drivers, this class of device uses a generic HID driver which is common for all HID devices.

G. SUMMARY

This chapter described the background for the technologies relevant to this thesis and includes the Least Privilege Separation Kernel (LPSK) of the Trusted Computing Exemplar (TCX) project. It also provides an overview of the USB standards and the different HCI specifications that are used for USB implementations. Chapter III describes the issues that had to be considered when preparing to incorporate USB support into the prototype LPSK, including the system boot process, the detection of USB controllers, and the communication data structures for the UHCI controller. Chapter III ends with an evaluation of three open source USB implementations as candidates for porting to the LPSK.

III. USB IMPLEMENTATION ISSUES FOR THE LPSK

Knowledge of the various specifications for USB allows the USB design to be understood. But this is not sufficient for the implementation. Implementation of USB support in the LPSK requires a working knowledge of the LPSK architecture so that kernel changes can be made. This chapter serves to provide the reader with the relevant issues that need to be addressed to provide a USB driver in the LPSK. The chapter ends with an evaluation of open source USB implementations.

A. SYSTEM STARTUP PROCESS

Support for USB is first realized through the implementation of a bus device driver that operates in the kernel space. To understand how the bus device driver is loaded and initialized, it is necessary to understand the different stages of a system startup process before application processes are started. A system upon power-up or reset goes through four different initialization stages before USB devices can be detected. These stages are determined by where the operating instructions reside in the system, which are namely the Basic Input-Output System (BIOS), primary boot loader, secondary boot loader and kernel initialization [15].

Once the system startup sequence is initiated, the code residing in BIOS is executed first. This code is stored in flash memory and is typically characterized as “firmware.” It performs two main functions: the first is to conduct a Power-On Self-Test (POST) on the system hardware and the second function is to enumerate and initialize all supported hardware devices. Then the BIOS will read the Master Boot Record (MBR) from a bootable device determined by the boot sequence stored in the BIOS configuration setting and copies it into system memory. The BIOS then jumps to the code in memory to begin the next stage.

The MBR contains the primary boot loader and a partition table. The primary boot loader will scan through the partition table to look for the active partition from which to boot. Once the active partition has been found, its boot record will be read into system memory for execution. The code that is executed is known as the secondary boot loader

and is also known as the kernel loader as it will load the kernel for the selected OS to boot from. Another function of the secondary boot loader is to support multi-boot, meaning that more than one OS may be installed onto the same system so that the operator is able to select the OS to load during system startup. As such, it must recognize different file systems so that it is able to load the selected kernel initialization code from disk.

In the case of the prototype LPSK, a modified version of the Grand Unified Bootloader GRUB (Version 0.97) is used as both the primary and secondary boot loader. The kernel initialization code initializes all hardware devices, which would include the USB controllers and eventual USB devices. The USB controller needs to be initialized after the PCI controller because the USB controller is connected as a PCI device.

B. LEGACY USB SUPPORT IN BIOS

The PS/2 interface was developed in 1987 by IBM as a way to connect keyboards and mice with their PCs, but with the development of USB, the PS/2 interface has mostly disappeared. Most modern BIOSes provide some level of basic support for USB keyboards and mice through PS/2 emulation. In these cases, there will be an option for enabling or disabling “Legacy USB support” in the BIOS configuration menu. Table 2 shows the supportability of a USB keyboard in the different stages of a system startup process on a system without USB drivers and with no legacy USB support.

Stage	USB Keyboard
BIOS	Supported
GRUB	Not supported
Kernel	Not supported

Table 2. Effect of disabling “USB legacy support” in BIOS

As shown in Table 2, with legacy USB support disabled in the BIOS, a USB keyboard will not work in either GRUB or the kernel. For them to work, USB keyboard driver support must be provided in the respective component. In this thesis, the focus is on developing a USB bus driver in the kernel. To support USB in GRUB (i.e., using a USB keyboard to select the partition to boot from), similar USB functionality will need to be incorporated.

C. PCI LOGIC FLOW

Initialization and configuration of PCI devices are fully software driven. Each PCI device is required to provide 256 bytes of standard configuration registers. The types and amount of information contained in the configuration registers are dependent on the Header Type (bits 23–16 of Register 0x0C). Of interest to this thesis is Header Type 0x00 which is used for devices that are connected directly to the PCI bus (i.e., non-bridge). The layout of the standard PCI registers is shown in Table 3.

Register	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
0x00	Device ID		Vendor ID	
0x04	Status		Command	
0x08	Class code	Subclass	Prog IF	Revision ID
0x0C	BIST	Header Type	Latency Timer	Cache Line Size
0x10	Base Address Register #0 (BAR0)			
0x14	Base Address Register #1 (BAR1)			
0x18	Base Address Register #2 (BAR2)			
0x1C	Base Address Register #3 (BAR3)			
0x20	Base Address Register #4 (BAR4)			
0x24	Base Address Register #5 (BAR5)			
0x28	Cardbus CIS Pointer			
0x2C	Subsystem ID		Subsystem Vendor ID	
0x30	Expansion ROM base address			
0x34	Reserved			Capabilities Ptr
0x38	Reserved			
0x3C	Max Latency	Min Grant	Interrupt PIN	Interrupt Line

Table 3. PCI configuration registers for header type 0x00 [From 16]

To access the PCI devices, there are two 32-bit I/O addresses used. Address 0xCF8 is the address for setting the register address to access, while 0xCFC is used for reading or writing of data. To uniquely identify each of these devices on a specific PCI bus, they have their own 32-bit configuration address which is composed of the bits shown in Table 4. A system can support up to 256 PCI buses (i.e., Bus Number = 8 bits), and each of these buses can support a maximum of 32 devices (i.e., Device Number = 5 bits). The bits of interest for this thesis are bits 8 through 23.

Bit(s)	31	30–24	23–16	15–11	10–8	7–2	1–0
Description	Enable Bit	Reserved	Bus Number	Device Number	Function Number	Register Number	00

Table 4. PCI 32-bit configuration address

For USB support, the initial processing requires iterating through the list of PCI devices that are detected during kernel initialization and then reading parts of the 256-byte configuration space (highlighted in Table 3) to determine which of these PCI devices are USB host controllers and to retrieve from the configuration space for each of the controller devices, information required for configuring the controllers. Based on this requirement, the Enable Bit is set to “1” while the reserved bits are set to “0.” The composition of Function, Device and Bus number will uniquely identify a specific function within a specific PCI device on a specific PCI bus.

The PCI register number is used to specify the location of the 32-bit register within the PCI configuration space. For example, to determine whether a PCI device is a USB host controller, the 32-bit register at 0x08 would be read to decode the Class code, Subclass and Programming Interface (Prog IF) fields. A value of 0x0C for Class code indicates that it is a “Serial Bus Controllers” while 0x03 in the Subclass further indicates that it is a USB host controller. Lastly, the Prog IF field will indicate the host controller interface, as shown in Table 5.

Prog IF	Description
0x00	Universal Host Controller Spec (UHCI)
0x10	Open Host Controller Spec (OHCI)
0x20	Enhanced Host Controller Spec (EHCI)
0x30	Extensible Host Controller Interface (xHCI)

Table 5. Prog IF field in PCI configuration registers

Other registers that are important for USB configuration are Interrupt PIN, Interrupt Line and BAR4. The first two are for configuring interrupts in the target USB host controller while BAR4 contains the base address of the device’s USB I/O registers.

D. UHCI OVERVIEW

This section describes the controller interface put forth by UHCI as an example of how a USB bus driver communicates with a USB controller. There are four types of data transfer in USB and they are namely, Isochronous, Interrupt, Control and Bulk. The characteristics of the different data transfers and the type of USB devices that use them are shown in Table 6. For example, USB keyboards use Control transfer for initialization and Interrupt transfer for the actual operations.

Type	Characteristics	Applicability
Isochronous	Constant, fixed rate data transfer for devices that continuously consume or produce data at a fixed rate	Audio CODECs (e.g., Microphones, Speakers)
Interrupt	Small, spontaneous data transfer for devices that do not continuously generate data but require quick response when it does	Human Interface Devices (e.g., Keyboards, Mice)
Control	Bursty data transfers generated on an ad_hoc basis by a host for transferring device control, status, and configuration information	All USB devices
Bulk	Guaranteed transfer of large amounts of data with a relaxed latency requirement (able to tolerate relatively large latency)	Mass Storage devices or Printers

Table 6. USB data transfer types

To communicate with a UHCI host controller, all requests must be in the form dictated by the data structure defined for UHCI [17]. There is a Frame List which is an array of 1024 entries. Each of these entries is equivalent to one USB transaction and can point to either a Transfer Descriptor (TD) or a Queue Head (QH). It will point to a TD if it is an isochronous operation. This TD is the beginning of a TD list which is ordered based on time of execution. For non-isochronous operations such as interrupt, control and

bulk transfers, the frame will point to a QH. This QH points to one or more TDs that form part of a transaction. For Interrupt transfer, there is only one QH with its associated TDs while Control and Bulk transfers consist of multiples QHs. A graphical illustration of the data structure is shown in Figure 6.

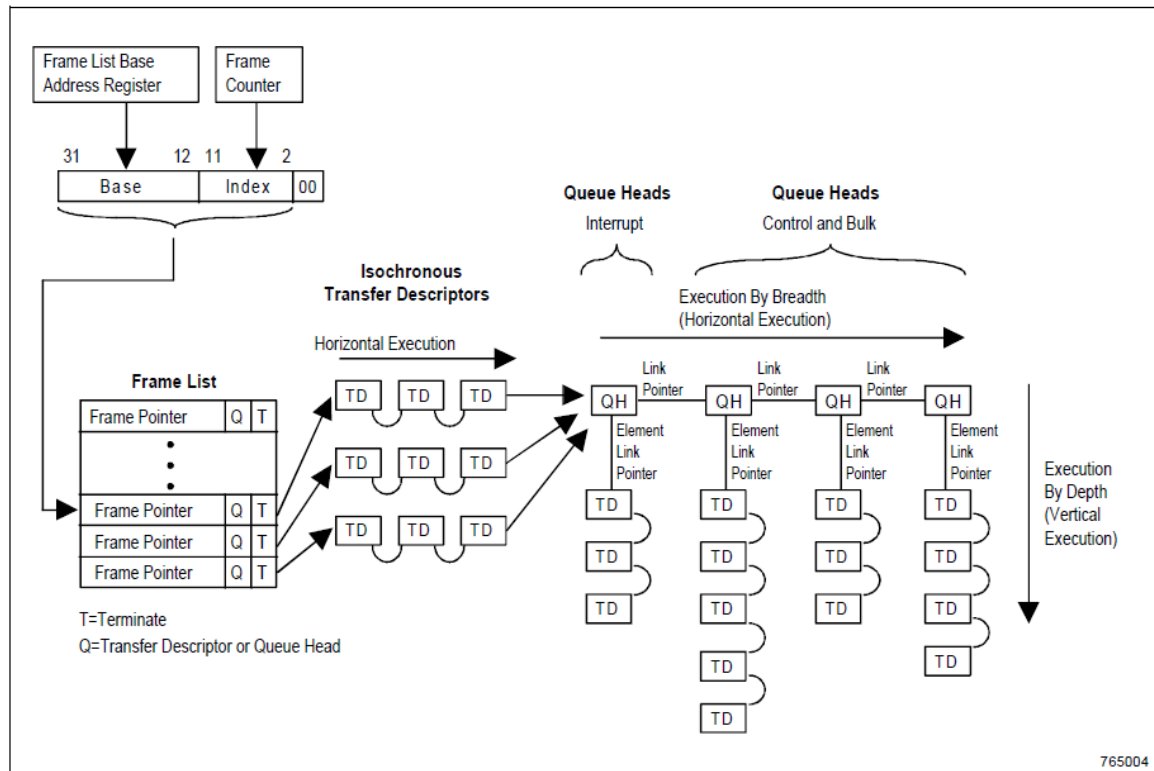


Figure 6. UHCI data structure [From 17]

Once the base address (Base) of an identified UHCI host controller is known, the I/O addresses for the various USB I/O registers can be computed. With this information, initialization and configuration of the host controller can be done. The different I/O registers for an UHCI controller are as shown in Table 7.

I/O Address	Mnemonic	Register Description	Size
Base	USBCMD	USB Command	16 bits
Base + 0x02	USBSTS	USB Status	16 bits
Base + 0x04	USBINTR	USB Interrupt Enable	16 bits
Base + 0x06	FRNUM	Frame Number	16 bits
Base + 0x08	FRBASEADD	Frame List Base Address	32 bits
Base + 0x0C	SOFMOD	Start of Frame Modify	8 bits
Base + 0x10	PORTSC1	Port 1 Status/Control	16 bits
Base + 0x12	PORTSC2	Port 2 Status/Control	16 bits

Table 7. UHCI I/O registers

The USBCMD register is used for sending commands to the host controller. A write to the register will cause the associated command to be executed. Some of the commands that can be issued are resetting the host controller and switching to debug mode.

The USBSTS register is used for checking the state of the host controller and it includes any pending interrupts that have occurred. When an error or interrupt occurs in the host controller while it is processing a USB transaction, the respective error or interrupt bit in this register will be set.

The USBINTR register is used for masking of interrupts so that they will not be reported. Interrupts that are non-fatal can be masked, including those triggered by timeout and checksum (CRC) errors.

The FRBASEADD and FRNUM registers are used by the software to inform the host controller of the location where the pending USB transaction information can be found. FRBASEADD indicates the starting address of the Frame List in the system memory while FRNUM indicates the current offset in the Frame List. The SOFMOD register is used for controlling the timing used for processing of the frames and is dependent on the frequency of the system clock.

Each UHCI host controller supports two virtual connections that are referred to as ports (not to be confused with physical USB ports) that can each be connected to one USB device or hub. To access the devices or hubs that are connected to the ports, PORTSC1 and PORTSC2 registers are used. These two registers can be used to check the status of the ports (e.g., disabled or suspended) or whether a device is connected to it. For controlling, the respective bit can be set to resume a port from a suspend state or for resetting the port.

In most PCs, an internal USB hub within the system is connected to virtual port PORTSC2 to increase the number of USB devices that can be connected. The first device that is physically plugged into a USB bus is connected to virtual port PORTSC1. If there is more than one device that is physically plugged into a USB bus, then all additional devices will be connected to the hub that is connected to virtual port PORTSC2.

E. OPEN SOURCE IMPLEMENTATIONS

In this thesis, an initial attempt to implement a USB bus driver from scratch was not successful as there is not much documentation available besides the official published USB specification. Information found on the Internet and books that are published on USB driver development focus on developing USB device drivers to support new devices in commonly used OSs such as Windows and Linux. Substantial source code analysis of existing implementations would have been required to write a USB bus driver from scratch.

To jumpstart the work in this thesis, a small search was then conducted to look into some available open source USB driver implementations that could potentially be ported to the LPSK. The key objective when conducting this search was to find source code with a small footprint that could be analyzed within a short period of time. This ruled out the USB implementation in Linux and other well-known open source OSes, where the USB code is many thousands of lines. Three less-known implementations were evaluated as possible porting candidates: Quest OS, Visual OS and Sea BIOS (previously known as Legacy BIOS).

Quest OS:

This OS evolved from a research project undertaken by Boston University in 2005 [18]. It was designed with efficient scheduling in mind. The source code, which can be found at <http://questOS.github.com/quest>, is dependent on GCC (GNU Compiler Collection). In evaluating the USB implementation in Quest OS, it was found that support for USB is limited to UHCI and devices such as mass storage, network, serial port and video camera. Keyboard support is provided in the form of PS/2 with no native HID support for USB. Because one of the development goals of Quest OS was to keep the footprint small, the development is not object-oriented (OO). There is very little documentation, which includes limited source code comments.

Visual OS:

Also known as Visopsys, Visual OS is an alternative OS for the PC that was undertaken by a hobbyist programmer in 1997 [19]. It was designed to be small and fast with support for a graphical based user interface. The source code, which can be found at <http://visopsys.org/download/index.php>, is dependent on GCC. After evaluating the USB implementation in Visual OS, it was found that support for USB is limited to UHCI and devices such as mass storage, keyboard and mouse. The implementation of the USB modules is OO based with modularity in design. There is a tight coupling to a kernel linked list that is used for maintaining information for all PCI devices. Documentation exists in the form of comments within the source code.

Sea BIOS:

Sea BIOS is an open source implementation of an x86 BIOS and is the default BIOS for Kernel-based Virtual Machine (KVM) from Red Hat, Inc [20]. The source code, which can be found at <http://www.linuxtogo.org/~kevin/SeaBIOS/>, is dependent on GCC. After evaluating the USB implementation in Sea BIOS, it was found that it supported all three different HCIs required for compliance with the USB 2.0 standard (i.e., UHCI, OHCI and EHCI) and it supports devices such as mass storage, keyboard and mouse. There are numerous comments in the source code and no tight coupling to data structures outside of the USB implementations.

Conclusion:

A summary of the USB support provided by the three open source OSes evaluated above is shown in Table 8. From the Table, one can observe that Sea BIOS provides a more comprehensive USB implementation. In addition, the Sea BIOS code is more portable, as described above. Therefore, the USB code base of Sea BIOS code was selected as the code to port to the LPSK.

OS	HCI supported			Device Class supported	
	UHCI	OHCI	EHCI	HID	Others
Quest OS	Yes	No	No	No	Mass Storage, Network, Serial Port and Video
Visual OS	Yes	No	No	Yes	Mass Storage
Sea BIOS	Yes	Yes	Yes	Yes	Mass Storage

Table 8. USB supports in open source OSes evaluated

F. SUMMARY

This chapter described the issues that surfaced after understanding the LPSK environment, how UHCI controllers and USB devices are detected, and related USB specifications. The results from the evaluation of three open source USB implementations were presented, which resulted in the selection of Sea BIOS as the porting candidate for USB support in the LPSK.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PORTING

This chapter describes how a partial USB bus driver was ported to the LPSK. Specifically, it describes the technical details about how the USB modules in Sea BIOS OS were ported to the LPSK.

The chapter begins with a description of the development environment used for the porting effort. This is followed by a discussion of LPSK modifications required to support USB devices. Then the Sea BIOS port is described along with problems encountered and resolved during this process.

A. DEVELOPMENT ENVIRONMENT

A version of the Open Watcom C/C++ compiler is required for the development of the LPSK. The Open Watcom compiler does not support some of the compiler directives, attributes and macros that are supported by the GNU Compiler Collection (GCC) that is used in most Linux developments. GCC is often the expected compiler for open source projects, including all three of the open source USB implementations evaluated in this thesis (Quest OS, Visual OS and Sea BIOS). Open Watcom is used for the LPSK because it supports the Large Memory Model, which is no longer supported by GCC. Changes to the source code for the related USB modules in Sea BIOS were required to overcome the compiler differences, and are covered in more detail in the subsequent sections.

For the purpose of development and testing, two different machines are used. One machine is setup for development with the Linux OS with a development environment based on Open Watcom C/C++ compiler. In addition, to aid in the review of source code for both the LPSK and open source USB implementations, source code analysis tools such as Source Navigator (version 4.4) were installed on the development machine for this thesis. The Source Navigator software allows the relationships between functions to be traced more easily and is also used for editing of source code.

The other machine is for the purpose of testing and is setup with multi-boot capability using GRUB. Two bootable partitions are installed and configured in this test

machine; one is a Linux OS, and the other one contains the LPSK. The Linux OS is used to connect to the development machine to download and install new LPSK binaries for testing. Actual testing of a new LPSK is then done by rebooting the test machine and selecting the LPSK from the boot menu.

To minimize the requirement for physical hardware, an alternative setup can be used based on VMware. In this case, both the development and testing systems would exist in the form of virtual machines on that same host.

The LPSK currently does not have any specialized debugging tools for kernel development. For this work, functions for printing out values or messages on the screen were used for debugging. To capture or view the displayed information, software delays were inserted into the source code. In inserting delays, it should be noted that delays could not be placed within time-critical code sections, as this would have altered the behavior of the system. Screen captures were also used to save an image of debugging output for subsequent analysis.

To verify that information retrieved from the USB devices was correct, the Linux “lsusb” command was used after booting the Linux partition on the test machine. This command with the “-v” parameter (i.e., verbose) was used for displaying information about USB buses on a system and the details (in term of descriptors) for USB devices that are connected to them.

B. LPSK MODIFICATIONS FOR USB SUPPORT

To incorporate USB support into the LPSK, it was necessary to understand the current software architecture of the prototype LPSK: specifically, how PCI devices are detected and how their information can be retrieved. In the LPSK source code, it was found that the code that detects and initializes PCI devices resides in the *pci_init()* function. This function is called by the *config_devices()* function during kernel initialization, which then allows other device-specific modules to be initialized by *config_devices()*, such as calling *nic_init()* (for network card) and *ide_init()* (for IDE devices). Without changing the existing architecture, an *usb_init()* function was added for detecting and initializing USB controllers and devices.

To retrieve the PCI information for devices connected to the PCI bus, two existing PCI-related functions were used. The first function, *pci_get_instance()* was used to return the address of the USB controller, and the *pci_get_type0_header()* function was used to retrieve the type 0 header (Refer to Table 3 for details), which is then decoded to extract the relevant USB information needed for downstream processing.

C. SEABIOS USB IMPLEMENTATION

The USB implementation in Sea BIOS supports the following USB host controllers and classes of USB devices (Refer to Appendix A for information on the different classes).

Host Controller Interfaces:

1. UHCI
2. OHCI
3. EHCI

Classes of USB devices:

1. HID (Human Interface Device)
2. Mass Storage
3. Hub

In Sea BIOS, the USB implementation is based on a layered approach. There is a main USB module (filename: “*usb.c*”) which relays the calls to the respective HCI modules (e.g., “*usb-uhci.c*,” “*usb-ohci.c*” or “*usb-ehci.c*”) depending on the controller type detected (Refer to Table 5 for details). Depending on the class of the USB device that the HCI modules is handling, the respective USB class module is invoked (e.g., “*usb-hid.c*,” “*usb-misc.c*” or “*usb-hub.c*”).

The focus of this thesis was the UHCI (i.e., “*usb-uhci.c*”) and Hub support (i.e., “*usb-hub.c*”). These allow the detection and retrieval of USB device information both directly and through a USB hub.

D. PROBLEM RESOLUTION

With an understanding of both the LPSK PCI architecture and USB implementation in Sea BIOS, the related code in Sea BIOS was modified and ported to the LPSK. A number of issues surfaced along the way and this section highlights the different types of problems encountered and how they were resolved.

The first category of problems encountered was caused by using a different compiler than expected, as described earlier. To make the Sea BIOS code compile properly with Open Watcom C/C++ compiler, a number of changes to the Sea BIOS code were required to overcome unsupported features.

Compiler Directives and Attributes:

There are a number of compiler directives and attributes used in the Sea BIOS code but not supported by the Open Watcom C/C++ compiler. Some of these can be safely ignored while equivalent features had to be sought for others.

An example is the “noinline” attribute, shown below:

```
int noinline my_function(void) {  
    // function variables and codes  
}
```

The “noinline” attribute is used in test stubs to prevent the compiler from attempting to optimize the function and not calling it when there is no useful executable code (i.e., when the code evaluates to a constant). In the case of the LPSK, the attribute can be safely removed as there is no requirement for use of test stubs:

```
int my_function(void) {  
    // function variables and codes  
}
```

The way one indicates the need for a packed data structure is different between the two compilers. The following is an example of how the original Sea BIOS code indicated a packed structure:

```
struct usb_config_descriptor {  
    u8  bLength;  
    u8  bDescriptorType;  
    // other declaration  
} PACKED;
```

The above use of PACKED had to be replaced with Open Watcom directives, as shown below:

```
#pragma pack(push, 1)
struct usb_config_descriptor {
    u8  bLength;
    u8  bDescriptorType;
    // other declaration
};
#pragma pack(pop)
```

Variable Declarations:

The GCC compiler allows a variable to be defined anywhere within a function. In the case of Open Watcom C/C++ compiler, all variables must be declared at the start of the function prior to any executable code or statement, or at the start of an explicit code block. This issue was easily overcome by identifying all the misplaced variables within each function and moving them to the beginning of the function code.

Pointer Arithmetic:

The GCC compiler allows pointer arithmetic to be performed on void pointers, apparently assuming that a void pointer is equivalent to a pointer to a character. In the case of Open Watcom C/C++ compiler, a compiler error will occur if that coding convention is used. The cause of the error is because the compiler does not know the exact type to be used for the arithmetic operation. To resolve this error, type casting to a character pointer was used.

For example, the code that looked like:

```
struct my_device* target;
int size;
// initialize my_device and size
target = (void*)target + size;
```

was changed to:

```
struct my_device* target;
int size;
// initialize my_device and size
target = (void*)((char*)target + size);
```


Specialized Macro:

The Sea BIOS code uses the *container_of()* macro, which was based upon a macro with the same name from the Linux kernel. A detailed explanation of what this macro does can be found at Linux Kernel Monkey Log website [21]. This macro in its original form cannot work in the Open Watcom C/C++ compiler environment. An equivalent macro, which does not perform type checking, was used for the LPSK.

Macro used in Sea BIOS:

```
#define container_of(ptr, type, member) ({  
    const typeof( ((type *)0)->member ) *__mptr = (ptr);  
    (type *) ( (char *)__mptr - offsetof(type,member) );})
```

Macro used in LPSK:

```
// modified from container_of of linux, no type checking here  
#define container_of(ptr, type, member) (type *) ((char *) (ptr) - \  
    offsetof(type, member))
```

Unsupported Inline Assembly Code:

Some functions in the Sea BIOS code are implemented using inline assembly code. Although inline assembly is support by the Open Watcom compiler, it requires a different syntax than what is required by GCC. Instead of translating the syntax, its C code equivalent was used for the LPSK.

Inline assembly function used in Sea BIOS:

```
// __fls - find last (most-significant) bit set  
// Note fls(0) = 0, fls(1) = 1, fls(0x80000000) = 32.  
static inline u32 __fls(u32 word)  
{  
    __asm("bsr %1,%0"  
        : "=r" (word)  
        : "rm" (word));  
    return word;  
}
```

C equivalent function used in LPSK:

```
static inline u32 __fls(u32 x)  
{  
    int r = 32;  
  
    if (!x)  
        return 0;
```

```

    if (!(x & 0xffff0000u))
    {
        x <<= 16;
        r -= 16;
    }
    if (!(x & 0xff000000u))
    {
        x <<= 8;
        r -= 8;
    }
    if (!(x & 0xf0000000u))
    {
        x <<= 4;
        r -= 4;
    }
    if (!(x & 0xc0000000u))
    {
        x <<= 2;
        r -= 2;
    }
    if (!(x & 0x80000000u))
    {
        x <<= 1;
        r -= 1;
    }
    return r;
}

```

The second category of problems encountered resulted from the architectural differences between Sea BIOS and the LPSK. To address some of these problems, new functions were created for the LPSK.

Physical and Virtual Memory Mapping:

Device drivers are written to use physical memory addresses. When a device driver needs to pass an address to a device, such as a buffer address, the driver must use a physical address. Software, on the other hand, uses the concept of a pointer, which is almost always not connected to a physical address. It is the responsibility of the driver to convert pointers to physical addresses, and vice versa, when interacting with a device.

In Sea BIOS, mapping between Physical and Virtual Memory can be achieved by simply type casting the memory address to an unsigned integer or void pointer. This is made possible through the creation of a single segment that starts at the physical offset of

0. The LPSK, on the other hand, makes greater use of segmentation, such that there is no direct correspondence between physical and virtual addresses. Instead, the LPSK provides the *kmem_phys()* function to translate a virtual address (i.e., a pointer) to a physical address.

To translate a physical memory address to a virtual memory address in the LPSK, a sequence of function calls must be used. First, the physical memory must be associated with a segment. In the case of the USB porting effort, the association between the physical memory and its associated segment was saved in the form of a selector when the segment was created. Second, using the selector as an input, the *get_base()* function is used to obtain the base address of the segment. Third, the segment offset is calculated by subtracting the base address from the physical address. Finally, giving the selector and offset as inputs, the virtual address in the form of a pointer can be derived using the *make_ptr()* function. A new function call *kmem_virt()* that encapsulates the above steps was created for the LPSK to facilitate the USB porting effort.

In porting the Sea BIOS code to the prototype LPSK, there are numerous instances where translation between a physical memory address and a virtual memory address are required. If any step of the address translation is incorrect, the LPSK will either crash or fail to work. When it crashed, the offending code was usually identified using the crash dump information displayed on the screen by the kernel. Unfortunately, it was very difficult to trace memory problems when the code failed to work properly because the debugging process involved tracing the values for all the elements (i.e., Framelist, QHs & TDs) in the UHCI data structure [17] as shown in Figure 6. In displaying these values for debugging, care had to be used to ensure that the delay introduced to display the values for tracing did not alter the behavior of the time-sensitive USB code. Of all the porting problems that were encountered, it was this debugging of memory problems that took the most time, by far.

Getting Elapsed Time in milliseconds:

Most of the USB operations and transactions are time critical. Software delays on the order of milliseconds (ms) may be added to the operations and transactions so that the

USB hardware timing requirements can be met. The LPSK provides an internal delay function called *clock_sleep()* that accepts an input parameter in the form of ms. Because this function blocks (i.e., will not return until it is completed), it cannot be used to calculate the elapsed time required for USB operations. To address this problem, another function called *get_time_elapsed_ms()* was created. This function returns the elapsed time in ms since system boot up. With this function, it is possible to create a timeout in term of ms.

Multi-threading:

The Sea BIOS code supports multi-threading and synchronization mechanisms such as mutex are used extensively throughout the code. In the case of the LPSK, since program execution is serialized, all the synchronization mechanisms are ignored (i.e., removed).

E. SUMMARY

This chapter described the work done in porting some of the USB modules (i.e., USB bus driver, UHCI driver and Hub driver) from the Sea BIOS to the LPSK. An overview of the development environment used for this work was presented. The various issues encountered in the porting work were covered together with their solution. In the next chapter, the details on the tests that were conducted on the USB modules in the LPSK are discussed.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING

Once the Sea BIOS's USB code was ported to the LPSK, testing was needed to ensure that the new USB modules worked correctly. This chapter covers the scope of testing done on the modified LPSK and provides the details required to replicate the tests.

A. TEST SETUP

For the purpose of testing the newly incorporated USB modules, there are two testing configurations used. The main difference between the two configurations is the use of VMware in one of the testing configurations. The hardware and software configurations for the two testing setups are as listed in Tables 9 and 10.

Hardware Configuration:	
Processor	AMD Phenom™ II P820 Triple-Core Processor 1.80 GHz
Installed Memory	4.0 GB
Software Configuration:	
Host OS	Windows 7 Home Premium (64-bit)
System Patch	Service Pack 1
VMware Version	VMware® Workstation (7.1.4 build-385536)
Boot Loader	GRUB 0.97 (modified to boot the LPSK)
LPSK	Modified Version 0.13

Table 9. Testing configuration with VMware

Hardware Configuration:	
Processor	Intel ® Atom TM CPU N280 1.66GHz
Installed Memory	2.0 GB
Software Configuration:	
Host OS	Not Applicable
System Patch	Not Applicable
Boot Loader	GRUB 0.97 (modified to boot the LPSK)
LPSK	Modified Version 0.13

Table 10. Testing configuration without VMware

Besides the two system configurations for providing the LPSK environment, for the purpose of testing the newly incorporated USB modules, a mixture of USB keyboards and mice was used. The manufacturer and model of the devices tested are as follows:

Keyboards:

1. DELL / RT7D10
2. DELL / SK-8115

Mice:

1. DELL / M-UK DEL3
2. Unbranded / USB Optical Mouse

B. TEST PROCEDURES

Two types of checks are used to validate the correctness of the code implemented. First, it is necessary to ensure the correct detection of the USB devices during the system startup process and second, one must ensure that data retrieved by the developed USB software is correct.

Part I of Preparation (Displaying information to be validated):

To facilitate the tests, the USB modules are coded such that the information necessary for the validation of the results is displayed on the screen with an inserted delay so that the information can be captured by the tester before it scrolls off the screen.

A sample code snippet for this purpose is shown below:

```
void print_device_descriptor(struct usb_device_descriptor dinfo)
{
    printf_lstr("*** DEVICE DESCRIPTOR ***\n");
    printf_int("bLength          = %d\n", dinfo.bLength);
    printf_int("bDescriptorType    = %d\n", dinfo.bDescriptorType);

    if (dinfo.bDescriptorType != USB_DT_DEVICE)
    {
        printf_lstr("* Abort Parsing. Incorrect descriptor type. *\n");
        return;
    }
    if (dinfo.bLength != sizeof(struct usb_device_descriptor))
    {
        printf_lstr("* Abort Parsing. Incorrect descriptor size. *\n");
        return;
    }
    printf_int("bcdUSB              = 0x%x\n", dinfo.bcdUSB);
    printf_int("bDeviceClass        = %d\n", dinfo.bDeviceClass);
    printf_int("bDeviceSubClass     = %d\n", dinfo.bDeviceSubClass);

    // Additional code here are not shown

    sleep(10);
}
```

In the above code, the USB device information contained in the structure “dinfo” is dumped to the screen using the *printf_lstr()* and *printf_int()* functions. The *sleep()* function at the end inserts a delay of 10 seconds to give the tester sufficient time to capture the results (either by screen capture or other manual means). This time can be arbitrarily increased and the code recompiled, if more time is needed.

Part II of Preparation (Gathering reference information):

To verify the correctness of the output from the modified LPSK, there must exist a reference set of results to validate against. The LPSK system resides on a PC (both virtual and physical) that is setup with multi-boot capability. The target test PC is built from an archived set of instructions managed through the CISR group. The target PC can

boot either a Linux OS or the LPSK. For testing purposes, the Linux OS is used to gather the reference set of results for the USB devices by using the native “lsusb” command. The parameter “-v” is specified to get the details necessary for validating the LPSK output. The sequence of steps is as follows:

- i) Boot into Linux OS on the target system.
- ii) Logon to the system as root.
- iii) Connect the test USB Keyboards and Mice to the system.
- iv) Execute the command “*lsusb -v > usbdevices.txt*”
- v) Repeat step (iii) and (iv) with a different output filename as necessary to gather information on all USB devices that need to be tested.
- vi) Copy all the output files (e.g., “*usbdevices.txt*”) to another system for processing and printing.
- vii) The output files produced in step (iv) contain the descriptors for all the USB devices and hubs found when the “*lsusb*” command was executed. The expected output for keyboards is as follows, where the *bInterfaceProtocol* field has a value of 1:

Device Descriptor:

<i>bLength</i>	18
<i>bDescriptorType</i>	1

...

Configuration Descriptor:

...

Interface Descriptor:

...

<i>bInterfaceClass</i>	3	<i>Human Interface Devices</i>
<i>bInterfaceSubClass</i>	1	<i>Boot Interface Subclass</i>
<i>bInterfaceProtocol</i>	1	<i>Keyboard</i>

...

Endpoint Descriptor:

...

- viii) For mice, the output is similar to keyboards except for the *bInterfaceProtocol* field which has a value of 2 to indicate a mouse device. For the purpose of testing, only the keyboard and mouse information are of interest.
- viii) Filter out the information from the output files for those devices that are not relevant to the test (i.e., those that did not match the output formats for keyboards and mice).

- x) The complete list of information for all the USB devices used for the tests in this thesis can be found in Appendix B.

Part III of Preparation (Configuring VMware):

For the VMware-based target PC, besides physically connecting the USB devices, two additional steps are required to logically connect the devices to the particular instance of LPSK that is undergoing the tests. These two steps are as follows:

- 1) Enable display of HID devices in VMware

By default, VMware does not display the connection status for USB HID devices. Hence, it is necessary to edit the particular instance of the LPSK to “Show all USB input devices.” The sequence of steps to enable the display of HID devices in VMware are as follows:

- i) Select the instance of the LPSK virtual machine that is used for testing.
- ii) Click on the “Edit virtual machine settings” in the “Commands” section.
- iii) In the pop up dialog box, select “USB Controller.”
- iv) Check the box for “Show all USB devices” in the “Connections” section.

This step only needs to be performed once for each virtual machine. A screen capture of the sequence of steps for the above is shown in Figure 7.

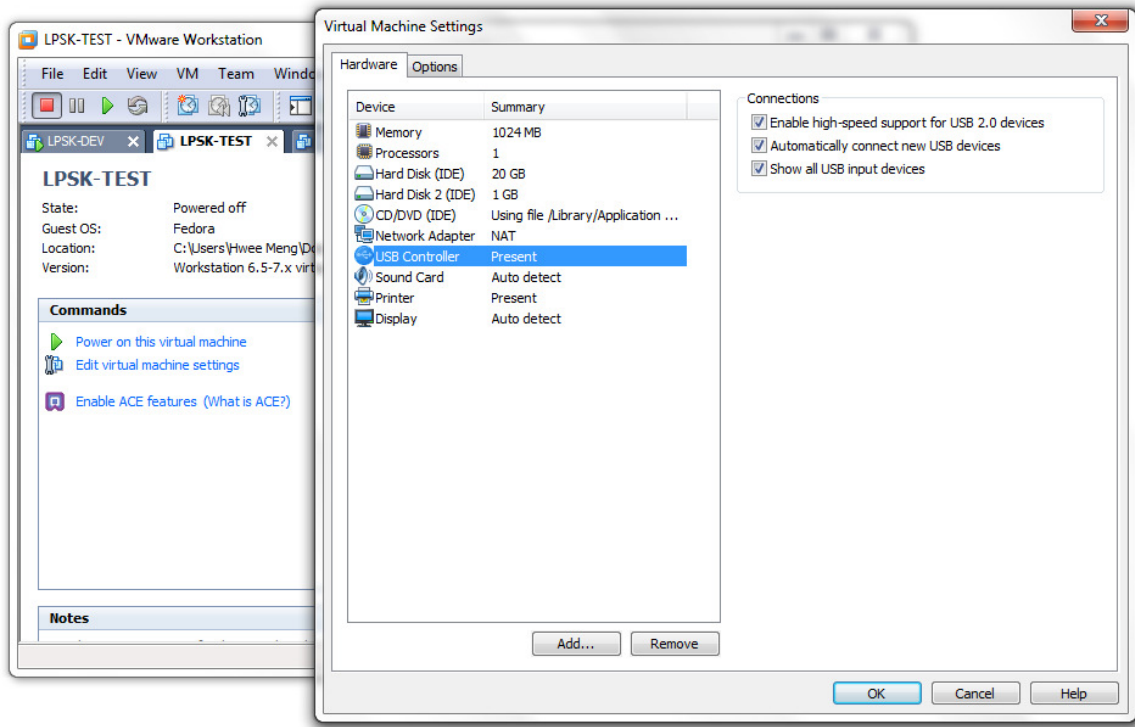


Figure 7. Show all USB devices

2) Connect the Keyboard and Mouse to the LPSK OS in VMware

By default, a device will be logically connected to the host OS when it is physically connected to a machine. To disconnect the device from the host and connect it to the virtual machine under test, the sequence of steps is as follows:

- i) Power on the LPSK virtual machine that is used for testing.
- ii) Click on the LPSK virtual machine and pause at the GRUB menu by pressing the “Down Arrow” key.
- iii) Re-focus to the host by pressing “CTRL + ALT” keys.
- iv) Select the “VM” menu and click on the “Removable Devices” option.
- v) The USB keyboard and mouse that are connected can be seen in the drop down list. Select the respective device and click on the “Connect (Disconnect from Host)” option, as shown in Figure 8.

- vi) Wait for the process to complete (by checking the respective icons at VMware status bar).
- vii) Click on and boot up the LPSK virtual machine.

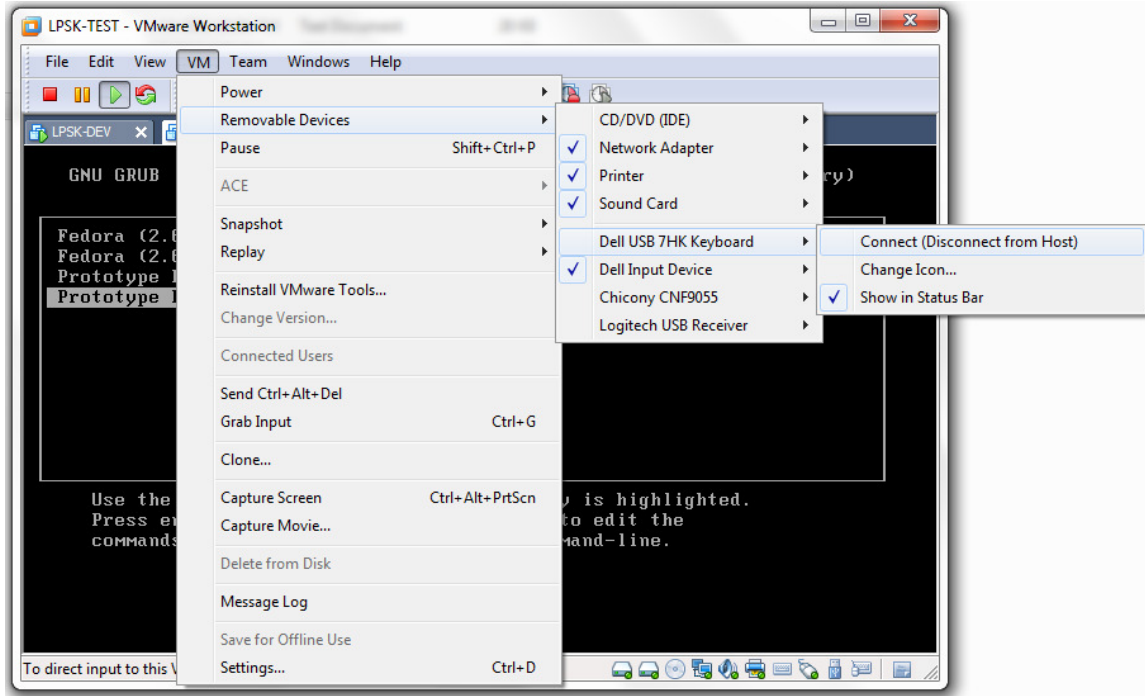


Figure 8. Connect (Disconnect from host)

Phase I of Test (Capturing information):

With the preparation work done, the actual tests for validating the correctness of the USB modules in the LPSK can be conducted. The first phase is to use the LPSK to gather the information for all the devices being tested. The sequence of steps involved is as follows:

- i) Connect the “Keyboard 1” and “Mouse 1” devices to the test machine
- ii) Power up the target machine
- iii) In the multi-boot menu, select the option to boot the LPSK
- iv) Either perform a screen capture or manually record the information for all the descriptors that are displayed. There should be information on the connected keyboard and mouse.

- v) Repeat steps (i) to (iv) for “Keyboard 2” and “Mouse 2.”
- vi) Repeat steps (i) to (v) on the virtual machine by performing the additional steps required for VMware.

Phase II of Test (Comparing information):

The second phase is to compare the information gathered in Phase I with the reference set of results (refer to Appendix B) gathered as part of preparation. The sequence of steps involved is as follows:

- i) From the captured information in Phase I, find the Device descriptor with the idVendor and idProduct fields that match the values for the Device descriptor of “Keyboard 1” in Appendix B.
- ii) Compare the rest of the Device descriptor with the set of results for “Keyboard 1” in Appendix B.
- iii) Likewise, compare the values for 1st Configuration descriptor, 1st Interface descriptor and 1st Endpoint descriptor.
- iv) Repeat steps (i) to (iii) for “Keyboard 2,” “Mouse 1” and “Mouse 2” using their respective values for idVendor and idProduct in their Device descriptor information.

C. TEST RESULTS

The tests described in Section B were conducted for the two testing configurations, namely, running the prototype LPSK on VMware as a virtual machine, and running the prototype LPSK on physical machines. In the tests where the LPSK executed in a VMware virtual machine, the ported USB modules were able to detect all connected USB devices. In contrast, when the LPSK was executed on physical machines, the USB modules failed to work. The test results described in this section only reflect the VMware tests.

Using the keyboards and mice described earlier, the test results using VMware are shown in Tables 11 through 14, which compare the USB output from the “lsusb” command and the LPSK. When the fields were compared, the factor deciding whether it

passed or failed is the actual value rather than the formatting used. For example, some of the fields are of BCD (Binary Coded Decimal) format. With a BCD value of “0x2910,” it will be displayed as “29.10” when it is formatted and output by the “lsusb” command.

Keyboard 1	Brand: DELL Model: RT7D10		
Device Descriptor	“lsusb”	LPSK	Result
bLength	18	18	Passed
bDescriptorType	1	1	Passed
bcdUSB	1.10	0x110	Passed
bDeviceClass	0	0	Passed
bDeviceSubClass	0	0	Passed
bDeviceProtocol	0	0	Passed
bMaxPacketSize0	8	8	Passed
idVendor	0x413c	0x413c	Passed
idProduct	0x2001	0x2001	Passed
bcdDevice	0.01	0x1	Passed
iManufacturer	3	3	Passed
iProduct	2	2	Passed
iSerial	0	0	Passed
bNumConfigurations	1	1	Passed
1st Configuration Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	2	2	Passed
wTotalLength	59	59	Passed
bNumInterfaces	2	2	Passed

bConfigurationValue	1	1	Passed
iConfiguration	2	2	Passed
bmAttributes	0xa0	0xa0	Passed
MaxPower	50mA	50mA	Passed
1st Interface Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	4	4	Passed
bInterfaceNumber	0	0	Passed
bAlternateSetting	0	0	Passed
bNumEndpoints	1	1	Passed
bInterfaceClass	3	3	Passed
bInterfaceSubClass	1	1	Passed
bInterfaceProtocol	1	1	Passed
iInterface	2	2	Passed
1st Endpoint Descriptor	“lsusb”	LPSK	Result
bLength	7	7	Passed
bDescriptorType	5	5	Passed
bEndpointAddress	0x81	0x81	Passed
bmAttributes	3	3	Passed
wMaxPacketSize	0x0008	8	Passed
bInterval	10	10	Passed

Table 11. Test result for Keyboard 1

Keyboard 2		Brand: DELL Model: SK-8115	
Device Descriptor	“lsusb”	LPSK	Result
bLength	18	18	Passed
bDescriptorType	1	1	Passed
bcdUSB	1.10	0x110	Passed
bDeviceClass	0	0	Passed
bDeviceSubClass	0	0	Passed
bDeviceProtocol	0	0	Passed
bMaxPacketSize0	8	8	Passed
idVendor	0x413c	0x413c	Passed
idProduct	0x2003	0x2003	Passed
bcdDevice	3.01	0x301	Passed
iManufacturer	1	1	Passed
iProduct	2	2	Passed
iSerial	0	0	Passed
bNumConfigurations	1	1	Passed
1 st Configuration Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	2	2	Passed
wTotalLength	34	34	Passed
bNumInterfaces	1	1	Passed
bConfigurationValue	1	1	Passed
iConfiguration	0	0	Passed

bmAttributes	0xa0	0xa0	Passed
MaxPower	70mA	70mA	Passed
1st Interface Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	4	4	Passed
bInterfaceNumber	0	0	Passed
bAlternateSetting	0	0	Passed
bNumEndpoints	1	1	Passed
bInterfaceClass	3	3	Passed
bInterfaceSubClass	1	1	Passed
bInterfaceProtocol	1	1	Passed
iInterface	0	0	Passed
1st Endpoint Descriptor	“lsusb”	LPSK	Result
bLength	7	7	Passed
bDescriptorType	5	5	Passed
bEndpointAddress	0x81	0x81	Passed
bmAttributes	3	3	Passed
wMaxPacketSize	0x0008	8	Passed
bInterval	24	24	Passed

Table 12. Test result for Keyboard 2

Mouse 1	Brand: DELL Model: M-UK DEL3		
Device Descriptor	“lsusb”	LPSK	Result
bLength	18	18	Passed
bDescriptorType	1	1	Passed
bcdUSB	1.10	0x110	Passed
bDeviceClass	0	0	Passed
bDeviceSubClass	0	0	Passed
bDeviceProtocol	0	0	Passed
bMaxPacketSize0	8	8	Passed
idVendor	0x413c	0x413c	Passed
idProduct	0x3200	0x3200	Passed
bcdDevice	29.10	0x2910	Passed
iManufacturer	1	1	Passed
iProduct	2	2	Passed
iSerial	0	0	Passed
bNumConfigurations	1	1	Passed
1 st Configuration Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	2	2	Passed
wTotalLength	34	34	Passed
bNumInterfaces	1	1	Passed
bConfigurationValue	1	1	Passed
iConfiguration	0	0	Passed

bmAttributes	0xa0	0xa0	Passed
MaxPower	50mA	50mA	Passed
1st Interface Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	4	4	Passed
bInterfaceNumber	0	0	Passed
bAlternateSetting	0	0	Passed
bNumEndpoints	1	1	Passed
bInterfaceClass	3	3	Passed
bInterfaceSubClass	1	1	Passed
bInterfaceProtocol	2	2	Passed
iInterface	0	0	Passed
1st Endpoint Descriptor	“lsusb”	LPSK	Result
bLength	7	7	Passed
bDescriptorType	5	5	Passed
bEndpointAddress	0x81	0x81	Passed
bmAttributes	3	3	Passed
wMaxPacketSize	0x0004	4	Passed
bInterval	10	10	Passed

Table 13. Test result for Mouse 1

Mouse 2		Brand: Unbranded Model: USB Optical Mouse	
Device Descriptor	“lsusb”	LPSK	Result
bLength	18	18	Passed
bDescriptorType	1	1	Passed
bcdUSB	2.00	0x200	Passed
bDeviceClass	0	0	Passed
bDeviceSubClass	0	0	Passed
bDeviceProtocol	0	0	Passed
bMaxPacketSize0	8	8	Passed
idVendor	0x15ca	0x15ca	Passed
idProduct	0x00c3	0xc3	Passed
bcdDevice	5.12	0x512	Passed
iManufacturer	0	0	Passed
iProduct	2	2	Passed
iSerial	0	0	Passed
bNumConfigurations	1	1	Passed
1st Configuration Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	2	2	Passed
wTotalLength	34	34	Passed
bNumInterfaces	1	1	Passed
bConfigurationValue	1	1	Passed
iConfiguration	0	0	Passed

bmAttributes	0xa0	0xa0	Passed
MaxPower	98mA	98mA	Passed
1st Interface Descriptor	“lsusb”	LPSK	Result
bLength	9	9	Passed
bDescriptorType	4	4	Passed
bInterfaceNumber	0	0	Passed
bAlternateSetting	0	0	Passed
bNumEndpoints	1	1	Passed
bInterfaceClass	3	3	Passed
bInterfaceSubClass	1	1	Passed
bInterfaceProtocol	2	2	Passed
iInterface	0	0	Passed
1st Endpoint Descriptor	“lsusb”	LPSK	Result
bLength	7	7	Passed
bDescriptorType	5	5	Passed
bEndpointAddress	0x81	0x81	Passed
bmAttributes	3	3	Passed
wMaxPacketSize	0x0004	4	Passed
bInterval	10	10	Passed

Table 14. Test result for Mouse 2

D. SUMMARY

This chapter described the test setup used for validating the correctness of the USB modules ported from the Sea BIOS OS to the LPSK. The test procedures detailed the steps required to replicate the tests. All the tests for the four USB devices used (i.e., two keyboards and two mice) were conducted successfully for the test configuration with VMware. There is no discrepancy in the outputs gathered from the LPSK when compared with the reference outputs taken from the Linux “lsusb” command. The same set of USB modules ported from Sea BIOS when tested in a non-VMware environment do not perform as expected: the system failed to detect the USB devices. The next chapter discusses the overall results of this thesis work and provides recommendations for future areas of work that may be undertaken to further enhanced the USB implementation in the LPSK.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

This chapter presents the overall result of the work done in this thesis based on test results presented in Chapter V and conclude with some recommendations for areas of USB implementation that can be looked into as part of future work.

A. RESULTS

The test results in Chapter V demonstrate that the USB functionalities ported to the prototype LPSK matches similar functions provided in Linux in a VMware environment. This provides the confidence in the correctness of the modules that are ported from Sea BIOS and adapted to compile under the Open Watcom C/C++ compiler. Specifically, the ported USB code detects USB devices and retrieves USB device information in the form of descriptors (e.g., Device, Configuration, Interface and Endpoint) for USB devices that are connected to an UHCI controller.

No changes were made to the overall LPSK software architecture. The design of the USB implementation in Sea BIOS allows the gradual migration of the remaining Sea BIOS USB modules into the LPSK. The lessons that were learned from the series of problems that were encountered (due to differences in compilers and software architectures) and their resolution will help in the subsequent work towards providing full USB support in the LPSK.

B. RECOMMENDATIONS FOR FUTURE WORK

There are several recommendations for future work, but the highest priority relates to the failure of the ported USB code in a non-VMware environment. Future effort needs to understand the differences between the LPSK environments when operating on a physical machine versus a virtual machine, with the aim to modify the USB code to work in both environments.

Among the four different transfer types in the USB specification, Control transfer is the most complicated (according to USB specification) and is validated to work correctly in this thesis. Isochronous transfer for UHCI is not implemented in Sea BIOS

while Interrupt transfer and Bulk transfer can be tested by porting the HID and Mass Storage modules respectively.

As a follow up for this thesis, it is recommended that a future effort be undertaken to port the HID (for keyboards and mice) module in Sea BIOS to the LPSK first. Bulk transfer is not crucial as most of today's USB devices that use bulk transfer are based on USB version 2.0, which will require additional effort to port the EHCI module.

Depending on the complexity of GRUB, some effort may also be undertaken to look into the feasibility of porting the validated USB modules into GRUB for a more complete USB support throughout the different stages in a system startup process.

In the course of work for this thesis, significant effort was spent in debugging and validating the outputs from the USB modules in the LPSK through the use of functions that display information on the screen (i.e., Video Display) and other functions that insert delays so that there is sufficient time to view or capture the outputs. To make kernel development more efficient so that time can be better utilized on the actual design and coding work, it is recommended that a mechanism for logging the output to a file or disk partition be explored. This feature will be invaluable for future enhancements of the LPSK in general, not just USB support.

APPENDIX A. USB CLASS CODE

The information in this appendix has been extracted without modification from http://www.usb.org/developers/defined_class

USB defines class code information that is used to identify a device's functionality and to nominally load a device driver based on that functionality. The information is contained in three bytes with the names Base Class, SubClass, and Protocol. (Note that 'Base Class' is used in this description to identify the first byte of the Class Code triple. That terminology is not used in the USB Specification). There are two places on a device where class code information can be placed. One place is in the Device Descriptor, and the other is in Interface Descriptors. Some defined class codes are allowed to be used only in a Device Descriptor, others can be used in both Device and Interface Descriptors, and some can only be used in Interface Descriptors. Table 15 shows the currently defined set of Base Class values, what the generic usage is, and where that Base Class can be used (either Device or Interface Descriptors or both).

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
10h	Interface	Audio/Video Devices
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FEh	Interface	Application Specific
FFh	Both	Vendor Specific

Table 15. USB class code

Base Class 03h (HID – Human Interface Device)

This base class is defined for devices that conform to the HID Device Class Specification found on the USB-IF website. That specification defines the usable set of SubClass and Protocol values. Values outside of that defined spec are reserved. These class codes can only be used in Interface Descriptors.

Base Class	SubClass	Protocol	Meaning
03h	xxh	xxh	HID device class

APPENDIX B. USB DEVICE INFORMATION FOR TESTING

This appendix contains all the device information from the Linux “lsusb -v” command for all the keyboards and mice used for testing, as listed below:

Keyboard 1	Make/Model: DELL / RT7D10																												
Device Descriptor	<table> <tr><td>bLength</td><td>18</td></tr> <tr><td>bDescriptorType</td><td>1</td></tr> <tr><td>bcdUSB</td><td>1.10</td></tr> <tr><td>bDeviceClass</td><td>0</td></tr> <tr><td>bDeviceSubClass</td><td>0</td></tr> <tr><td>bDeviceProtocol</td><td>0</td></tr> <tr><td>bMaxPacketSize0</td><td>8</td></tr> <tr><td>idVendor</td><td>0x413c</td></tr> <tr><td>idProduct</td><td>0x2001</td></tr> <tr><td>bcdDevice</td><td>0.01</td></tr> <tr><td>iManufacturer</td><td>3</td></tr> <tr><td>iProduct</td><td>2</td></tr> <tr><td>iSerial</td><td>0</td></tr> <tr><td>bNumConfigurations</td><td>1</td></tr> </table>	bLength	18	bDescriptorType	1	bcdUSB	1.10	bDeviceClass	0	bDeviceSubClass	0	bDeviceProtocol	0	bMaxPacketSize0	8	idVendor	0x413c	idProduct	0x2001	bcdDevice	0.01	iManufacturer	3	iProduct	2	iSerial	0	bNumConfigurations	1
bLength	18																												
bDescriptorType	1																												
bcdUSB	1.10																												
bDeviceClass	0																												
bDeviceSubClass	0																												
bDeviceProtocol	0																												
bMaxPacketSize0	8																												
idVendor	0x413c																												
idProduct	0x2001																												
bcdDevice	0.01																												
iManufacturer	3																												
iProduct	2																												
iSerial	0																												
bNumConfigurations	1																												
1 st Configuration Descriptor	<table> <tr><td>bLength</td><td>9</td></tr> <tr><td>bDescriptorType</td><td>2</td></tr> <tr><td>wTotalLength</td><td>59</td></tr> <tr><td>bNumInterfaces</td><td>2</td></tr> <tr><td>bConfigurationValue</td><td>1</td></tr> <tr><td>iConfiguration</td><td>2</td></tr> <tr><td>bmAttributes</td><td>0xa0</td></tr> <tr><td>MaxPower</td><td>50mA</td></tr> </table>	bLength	9	bDescriptorType	2	wTotalLength	59	bNumInterfaces	2	bConfigurationValue	1	iConfiguration	2	bmAttributes	0xa0	MaxPower	50mA												
bLength	9																												
bDescriptorType	2																												
wTotalLength	59																												
bNumInterfaces	2																												
bConfigurationValue	1																												
iConfiguration	2																												
bmAttributes	0xa0																												
MaxPower	50mA																												

1 st Interface Descriptor	bLength	9
	bDescriptorType	4
	bInterfaceNumber	0
	bAlternateSetting	0
	bNumEndpoints	1
	bInterfaceClass	3
	bInterfaceSubClass	1
	bInterfaceProtocol	1
	iInterface	2
1 st Endpoint Descriptor	bLength	7
	bDescriptorType	5
	bEndpointAddress	0x81
	bmAttributes	3
	wMaxPacketSize	0x0008
	bInterval	10

Table 16. Reference result for Keyboard 1

Keyboard 2	Make/Model: DELL / SK-8115
Device Descriptor	<div> <div>bLength</div> <div>18</div> </div> <div> <div>bDescriptorType</div> <div>1</div> </div> <div> <div>bcdUSB</div> <div>1.10</div> </div> <div> <div>bDeviceClass</div> <div>0</div> </div> <div> <div>bDeviceSubClass</div> <div>0</div> </div> <div> <div>bDeviceProtocol</div> <div>0</div> </div> <div> <div>bMaxPacketSize0</div> <div>8</div> </div> <div> <div>idVendor</div> <div>0x413c</div> </div> <div> <div>idProduct</div> <div>0x2003</div> </div> <div> <div>bcdDevice</div> <div>3.01</div> </div> <div> <div>iManufacturer</div> <div>1</div> </div> <div> <div>iProduct</div> <div>2</div> </div> <div> <div>iSerial</div> <div>0</div> </div> <div> <div>bNumConfigurations</div> <div>1</div> </div>
1 st Configuration Descriptor	<div> <div>bLength</div> <div>9</div> </div> <div> <div>bDescriptorType</div> <div>2</div> </div> <div> <div>wTotalLength</div> <div>34</div> </div> <div> <div>bNumInterfaces</div> <div>1</div> </div> <div> <div>bConfigurationValue</div> <div>1</div> </div> <div> <div>iConfiguration</div> <div>0</div> </div> <div> <div>bmAttributes</div> <div>0xa0</div> </div> <div> <div>MaxPower</div> <div>70mA</div> </div>
1 st Interface Descriptor	<div> <div>bLength</div> <div>9</div> </div> <div> <div>bDescriptorType</div> <div>4</div> </div>

	bInterfaceNumber 0 bAlternateSetting 0 bNumEndpoints 1 bInterfaceClass 3 bInterfaceSubClass 1 bInterfaceProtocol 1 iInterface 0
1 st Endpoint Descriptor	bLength 7 bDescriptorType 5 bEndpointAddress 0x81 bmAttributes 3 wMaxPacketSize 0x0008 bInterval 24

Table 17. Reference result for Keyboard 2

Mouse 1	Make/Model: DELL / M-UK DEL3
Device Descriptor	<div> <div>bLength</div> <div>18</div> </div> <div> <div>bDescriptorType</div> <div>1</div> </div> <div> <div>bcdUSB</div> <div>1.10</div> </div> <div> <div>bDeviceClass</div> <div>0</div> </div> <div> <div>bDeviceSubClass</div> <div>0</div> </div> <div> <div>bDeviceProtocol</div> <div>0</div> </div> <div> <div>bMaxPacketSize0</div> <div>8</div> </div> <div> <div>idVendor</div> <div>0x413c</div> </div> <div> <div>idProduct</div> <div>0x3200</div> </div> <div> <div>bcdDevice</div> <div>29.10</div> </div> <div> <div>iManufacturer</div> <div>1</div> </div> <div> <div>iProduct</div> <div>2</div> </div> <div> <div>iSerial</div> <div>0</div> </div> <div> <div>bNumConfigurations</div> <div>1</div> </div>
1 st Configuration Descriptor	<div> <div>bLength</div> <div>9</div> </div> <div> <div>bDescriptorType</div> <div>2</div> </div> <div> <div>wTotalLength</div> <div>34</div> </div> <div> <div>bNumInterfaces</div> <div>1</div> </div> <div> <div>bConfigurationValue</div> <div>1</div> </div> <div> <div>iConfiguration</div> <div>0</div> </div> <div> <div>bmAttributes</div> <div>0xa0</div> </div> <div> <div>MaxPower</div> <div>50mA</div> </div>
1 st Interface Descriptor	<div> <div>bLength</div> <div>9</div> </div> <div> <div>bDescriptorType</div> <div>4</div> </div>

	bInterfaceNumber 0 bAlternateSetting 0 bNumEndpoints 1 bInterfaceClass 3 bInterfaceSubClass 1 bInterfaceProtocol 2 iInterface 0
1 st Endpoint Descriptor	bLength 7 bDescriptorType 5 bEndpointAddress 0x81 bmAttributes 3 wMaxPacketSize 0x0004 bInterval 10

Table 18. Reference result for Mouse 1

Mouse 2	Make/Model: Unbranded / USB Optical Mouse
Device Descriptor	bLength 18 bDescriptorType 1 bcdUSB 2.00 bDeviceClass 0 bDeviceSubClass 0 bDeviceProtocol 0 bMaxPacketSize0 8 idVendor 0x15ca idProduct 0x00c3 bcdDevice 5.12 iManufacturer 0 iProduct 2 iSerial 0 bNumConfigurations 1
1 st Configuration Descriptor	bLength 9 bDescriptorType 2 wTotalLength 34 bNumInterfaces 1 bConfigurationValue 1 iConfiguration 0 bmAttributes 0xa0 MaxPower 98mA
1 st Interface Descriptor	bLength 9 bDescriptorType 4

	bInterfaceNumber 0 bAlternateSetting 0 bNumEndpoints 1 bInterfaceClass 3 bInterfaceSubClass 1 bInterfaceProtocol 2 iInterface 0
1 st Endpoint Descriptor	bLength 7 bDescriptorType 5 bEndpointAddress 0x81 bmAttributes 3 wMaxPacketSize 0x0004 bInterval 10

Table 19. Reference result for Mouse 2

LIST OF REFERENCES

- [1] S.R. Jr. Ames, M. Gasser and R.R. Schell, "Security kernel design and implementation: an introduction," *Computer*, vol. 16, no. 7, pp. 14–22, July 1983.
- [2] J. Anderson, "Computer security technology planning study," Air Force Elec. Syst. Div. Rep. ESD-TR-73–51, Oct. 1972.
- [3] J. Rushby, "The design and verification of secure systems," 8th ACM Symposium on Operating System Principles (SOSP), Pacific Grove, CA, Appears in *ACM Operating Systems Review*, vol. 15, no. 5, pp. 12–21, December 1981.
- [4] National Security Agency. "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness," Version 1.03, 29 June 2007.
- [5] Common Criteria for Information Technology Security Evaluation, Part 3: Security Assurance Requirements, Version 2.1, August 1999.
- [6] T.D. Nguyen, T.E. Levin, C.E. Irvine, "High Robustness Requirements in a Common Criteria Protection Profile," in *Proceedings of the 4th IEEE International Information Assurance Workshop*, pp.66–75, April 2006.
- [7] J. H. Saltzer and M.D. Schroeder, "The protection of information in computer systems," in *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, September 1975.
- [8] T. E. Levin, C. E. Irvine and T. D. Nguyen, "Least privilege in separation kernels," in *Proceeding of the International Conference on Security and Cryptography*, pp. 355–362, August 2006.
- [9] T. D. Nguyen, T. E. Levin and C. E. Irvine, "TCX project: high assurance for secure embedded systems," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 21–25, March 2005.
- [10] Howstuffworks.com, "System Bus vs. PCI Bus," <http://computer.howstuffworks.com/pci1.htm>. Last Accessed: Jan 2012.
- [11] PCIDatabase.com, "PCI Vendor and Device Lists," <http://www.pcidatabase.com>. Last Accessed: Jan 2012.
- [12] PCI Special Interest Group (PCI-SIG), "Conventional PCI 3.0 & 2.3: An Evolution of the Conventional PCI Local Bus Specification," <http://www.pcisig.com/specifications/conventional/>. Last Accessed: Jan 2012.

- [13] Intel, “Enhanced Host Controller Interface Specification for Universal Serial Bus,” Version 1.0, 12 Mar 2002.
- [14] USB Implementers Forum, Inc. (USB-IF), “Universal Serial Bus (USB) Device Class Definition for Human Interface Devices (HID),” Version 1.11, 27 Jun 2001.
- [15] IBM, “Inside the Linux boot process,”
<http://www.ibm.com/developerworks/linux/library/l-linuxboot/>. Last Accessed: Jan 2012.
- [16] OSDev.org, “PCI Device Structure,”
http://wiki.osdev.org/PCI#PCI_Device_Structure. Last Accessed: Oct 2011.
- [17] Intel, “Universal Host Controller Interface (UHCI) Design Guide,” Revision 1.1, March 1996.
- [18] Quest OS, “QUEST,” <http://questos.github.com/quest/>. Last Accessed: Jan 2012.
- [19] Visual OS, “Visopsys” <http://visopsys.org/a>. Last Accessed: Jan 2012.
- [20] Sea BIOS, “Sea BIOS,” <http://seabios.org/>. Last Accessed: Feb 2012.
- [21] Greg’s Log , “Linux Kernel Monkey Log,”
http://www.kroah.com/log/linux/container_of.html. Last Accessed: Jan 2012.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Paul C. Clark
Naval Postgraduate School
Monterey, California
4. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, California
5. Tan Hwee Meng
Defence Science & Technology Agency
Singapore